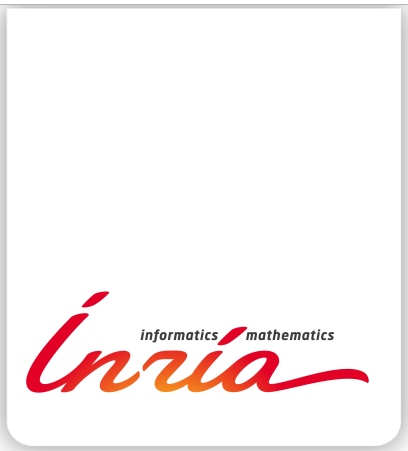
Resilience for Stencil Computations with Latent Errors

Aiman Fang, Aurélien Cavelan, Yves Robert and Andrew A. Chien

RESEARCH REPORT
N° 9042
March 2017
Project-Team ROMA

ISSN 0249-6399 ISRN INRIA/RR--9042--FR+ENG



Resilience for Stencil Computations with Latent Errors

Aiman Fang*, Aurélien Cavelan†, Yves Robert‡† and

Andrew A. Chien§*

Project-Team ROMA

Research Report n° 9042 — March 2017 — 30 pages

Abstract: Projections and measurements of error rates in near-exascale and exascale systems suggest a dramatic growth, due to extreme scale (10^9 cores), concurrency, software complexity, and deep submicron transistor scaling. Such a growth makes resilience a critical concern, and may increase the incidence of errors that "escape", silently corrupting application state. Such errors can often be revealed by application software tests but with long latencies, and thus are known as *latent errors*. We explore how to efficiently recover from latent errors, with an approach called application-based focused recovery (ABFR). Specifically we present a case study of stencil computations, a widely useful computational structure, showing how ABFR focuses recovery effort where needed, using intelligent testing and pruning to reduce recovery effort, and enables recovery effort to be overlapped with application computation. We analyze and characterize the ABFR approach on stencils, creating a performance model parameterized by error rate and detection interval (latency). We compare projections from the model to experimental results with the Chombo stencil application, validating the model and showing that ABFR on stencil can achieve a significant reductions in error recovery cost (up to 400x) and recovery latency (up to 4x). Such reductions enable efficient execution at scale with high latent error rates.

Key-words: resilience, fail-stop errors, multi-level checkpointing, optimal pattern.

* University of Chicago, USA
† Ecole Normale Superieure de Lyon and Inria, France
‡ University of Tennessee Knoxville, USA
§ Argonne National Laboratory, USA

RESEARCH CENTRE GRENOBLE – RHÔNE-ALPES

Inovallée
655 avenue de l'Europe Montbonnot
38334 Saint Ismier Cedex

Résilience pour des calculs de type “stencil” avec des erreurs latentes

Résumé : Les projections et mesures pour les systèmes exascale (10^9 coeurs) suggèrent une augmentation très importante du taux d'erreur. Une telle augmentation fait de la résilience un sujet critique, et risque d'aggraver l'impact des erreurs qui “s'échappent”, corrompant silencieusement la mémoire. Ces erreurs sont souvent détectées par des tests logiciels au niveau de l'application, mais avec une latence de détection importante, et sont donc connues sous le nom d'*erreurs latentes*. Nous explorons une approche appelée *application-based-focus-recovery*, ou ABFR, afin de relancer l'exécution efficacement, suit à une erreur. En particulier, nous présentons une étude de cas pour les applications de type stencil, montrant comment ABFR concentre les calculs de récupération où ils sont nécessaire, utilisant des tests et des élagages intelligents pour réduire les calculs de récupération, et permettre le recouvrement avec les calculs de l'application. Nous analysons et caractérisons l'approche ABFR pour les applications de type stencil, créant un modèle de performance paramétré par le taux d'erreur et l'interval de détection (la latence). Nous comparons les projections du modèle aux résultats expérimentaux avec l'application stencil Chombo, validant le modèle et montrant que ABFR permet d'obtenir une réduction significative du coût de récupération (jusqu'à 400x) et de la latence (jusqu'à 4x). De telles réductions de coût permettent de passer à l'échelle avec des taux d'erreurs latentes élevés.

Mots-clés : résilience, erreurs latentes, stencil, ABFR.

1 Introduction

Large-scale computing is essential for addressing scientific and engineering challenges in many areas. To meet these needs, supercomputers have grown rapidly in scale and complexity. They typically consist of millions of components [7], with growing complexity of software services [3]. In such systems, errors come from both software and hardware [28, 29]; both hardware-correctable errors and latent (or so-called silent) errors [10, 27] are projected to increase significantly, producing mean time between failure (MTBF) as low as a few minutes [9, 33]. Latent errors are detected as data corruption, but some time after their occurrence.

We focus on latent errors, that escape simple system level detection such as error-correction in memory, and can only be exposed by sophisticated application, algorithm, and domain-semantics checks [11, 26]. These errors are of particular concern, since their data corruption, if undetected and uncorrected, threatens the validity of computational (and scientific) results. Such latent errors can be exposed by sophisticated software level checks, but such checking is often computationally expensive, so it must be infrequent. We use the term "detection latency" to denote the time from error occurrence to detection, which may be 10^3 (thousands) to 10^9 (billions) of cycles. This delay allows corrupting a range of computation data. Thus, we detect the resulting data corruption, rather than the original error.

Checkpoint-Restart (CR) is a widely-used fault tolerance technique, where resilience is achieved by writing periodic checkpoints, and using rollback and recovery in case of failure. Rising error rates require frequent checkpoints for efficient execution, and fortunately new, low-cost techniques have emerged [10, 15]. Paradoxically, more frequent checkpoint increase the challenge with latent errors, as each checkpoint must be checked for errors as well. As a result not all checkpoints can be verified, and latent errors escape into checkpoints. Thus, improved checkpointing does not obviously help with latent errors. Keeping multiple checkpoints or using multi-level checkpointing systems have been proposed [4, 5, 25, 27, 30]; for latent errors, these systems search backward through the checkpoints, restarting, reexecuting, and retesting for error. Such iterated recovery is expensive, making development of alternatives desirable.

Algorithm-based fault tolerance (ABFT) exploits algorithm features and data structures to detect and correct errors, and can be used on latent errors. ABFT has been primarily developed for linear-algebra kernels [11, 17, 26, 31], including efficient schemes to correct single and double errors. However, each applies only to specific algorithms and data structures. Inspired by ABFT, we exploit application semantics to bound error impact and further localize recovery. Our central idea is to utilize algorithm dataflow and intermediate application states to identify potential root causes of a latent error. Diagnosing this data can enable recovery effort to be confined, reducing cost.

We exploit Global View Resilience (GVR) to create inexpensive versions of application states, and utilize them for diagnosis and recovery. In prior work [12, 13], GVR demonstrated that versioning cost is as low as 1% of total cost for frequent versioning under high error rates. A range of flexible rollback and forward recovery is feasible, exploiting convenient access to versioned state.

We propose and explore a new approach, application-based focused recovery (ABFR), that exploits data corruption detection and application data flow, to focus recovery effort on an accurate estimate of potentially corrupted data. In many applications, errors take time to propagate through data, so ABFR utilizes application structure to intelligently confine error recovery effort, and allow overlapped recovery. In contrast, global recovery does neither.

We apply this approach to a model application, stencil-based computations, a widely used paradigm for scientific computing, such as computation simulations, solving partial differential equations and image processing. We create an analytical performance model to explore the potential benefits of ABFR for stencil methods, varying dimensions such as error rate, error latencies and error detection intervals. The model enables us to characterize the advantages of ABFR across a wide range of system and application parameters. To validate the model, we perform a set of ABFR experiments, using the Chombo heat equation kernel (2-D stencil). The empirical results show that ABFR can improve recovery from latent errors significantly. For example, recovery cost (consumed CPU time) can be reduced by over 400-fold, and recovery latency (execution runtime) can be reduced by up to four-fold. Specific contributions of the paper include:

- A new approach to latent error recovery, algorithm-based focused recovery (ABFR), that exploits application data flow to focus recovery effort, thereby reducing the cost of latent error recovery;
- An analytical performance model for ABFR on stencil computations, and its use to highlight areas where significant performance advantages can be achieved;
- Experiments with the Chombo stencil computations, applying ABFR, both validating the model and demonstrating its practical application and effectiveness, reducing recovery cost by up to 400x, and recovery latency by up to 4x.

The remainder of the paper is organized as follows: Section 2 introduces the GVR library and stencil computations. In Section 3, we describe the ABFR recovery method, applied to stencil computations. Section 4 presents an analytical performance model for recovery, parameterized by error rate and detection interval (error latency). In Section 5, we present experiments with Chombo that validate the model, and provide quantitative benefits. Section 6 discusses classes of promising candidate applications of ABFR and limitations. Related work is presented in Section 7. Finally, we summarize

our work in Section 8, suggesting directions for future research.

2 Background

2.1 Global View Resilience (GVR)

We use the GVR library to preserve application data and enable flexible recovery. GVR provides a global view of array data, enabling an application to easily create, version and restore (partial or entire) arrays. In addition, GVR's convenient naming enables applications to flexibly compute across versions of single or multiple arrays.

GVR users can control where (data structure) and when (timing and rate) array versioning is done, and tune the parameters according to the needs of the application. The ability to create multi-version array and partially materialize them, enables flexible recovery across versions. GVR has been used to demonstrate flexible multi-version rollback, forward error correction, and other creative recovery schemes [19, 22]. Demonstrations include high-error rates, and results show modest runtime cost ($< 1\%$) and programming effort in full-scale molecular dynamics, Monte Carlo, adaptive mesh, and indirect linear solver applications [12, 13].

GVR exploits both DRAM and high bandwidth and capacity burst buffers or other forms of non-volatile memory to enable low-cost, frequent versioning and retention of large numbers of versions. As needed, local disks and parallel file system can also be exploited for additional capacity. For example, NERSC Cori [2] supercomputer provides 1.8 PB SSDs in the burst buffer, with 1.7 TB/s aggregate bandwidth (6 GB/s per node). The JUQUEEN supercomputer at Jülich Supercomputing Center [1] is equipped with 2 TB flash memory, providing 2 GB/s bandwidth per node. Multi-versioning performance studies on JUQUEEN [1] showed GVR is able to create versions at full bandwidth, demonstrating low cost versioning is a reality [20]. In this paper, GVR's low-cost versioning enables flexible recovery for ABFR.

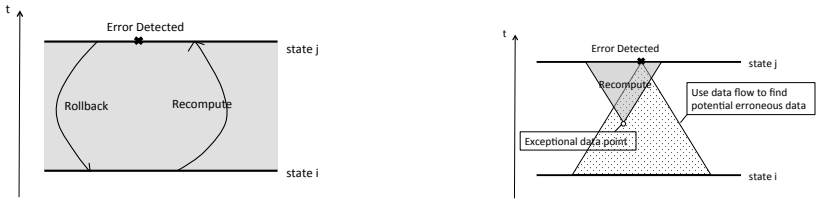
2.2 Stencils

Stencils are a class of iterative kernels that update array elements in a fixed pattern, called stencil. Stencil-based kernels are the core of a significant set of scientific applications [16, 21], and are widely-used in physical simulations, computational fluid dynamics, PDE solvers, cosmology, combustion, and image processing. Stencils involve computations across a set of 5-100 neighbors, with typical iterative structure as follows:

```
for k timesteps do
  - compute each element in array
    using neighbors in a fixed pattern
  - exchange the new value with neighbors
```

```
end
```

During execution, each process computes local elements and communicates with direct neighbors. The regular structure of stencils and their communication pattern suggest that errors take time to propagate to the whole data. Given error latency and location, we can use the communication pattern to identify potentially corrupted data and bound the recovery scope. We consider 5-point 2D stencil computations in subsequent sections, but the modeling and concepts can be extended in a straightforward fashion to higher dimensions and more complex stencils, see the extended version of this work for details (Appendix Section 8).



(a) Blind CR system recovery

(b) ABFR recovery based on application knowledge

Figure 1: Checkpoint Restart (CR) vs. Algorithm-based Focused Recovery (ABFR).

3 Algorithm-Based Focused Recovery (ABFR) Approach

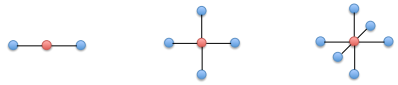
Many applications have regular, local data dependences or well-known communication patterns. Algorithm-based focused recovery (ABFR) exploits this knowledge to: (i) identify potentially corrupted data and focus recovery effort on a small subset (see Figure 1); and (ii) allow recovery to be overlapped, reducing recovery overhead and enabling tolerance of high error rates. In contrast, checkpoint-restart blindly rolls back the entire computation to the last verified checkpoint and recomputes everything.

ABFR is a type of ABFT method [26] that can be applied more generally. ABFR shares the ideas of overlapped, local recovery with [24], but extends them in scope and with sophisticated diagnosis. Specifically, ABFR's enables only the processes whose data is affected by errors to participate in the recovery process, and other processes to continue computation (overlapping recovery, subject to application data dependencies). By bounding error scope, ABFR saves CPU throughput, reducing recovery cost. Furthermore, overlapping recovery and computation can reduce runtime overhead significantly, enabling tolerance of high error rates.

In this paper, we describe an ABFR approach for stencil computations subject to latent errors. We assume that a latent error detector (or "error check") is available. Such detectors are application-specific and computationally expensive. In order to keep the model general, we make the following assumptions:

- The error detector has 100%[1] coverage, finding some manifestation whenever there is an error, but not precisely identifying all manifestations.
- The error check detects error manifestations in the data, namely, corrupted values and their locations.
- Because latent ("silent") errors are complex to identify, the detector is computationally expensive.[2]

As with other ABFT approaches, we utilize application semantics to design error detectors. Example detectors include: (i) temperature variation across timesteps within a threshold; (ii) one point within a range compared to its direct neighbors; (iii) average or total heat conservation, including fluxes; and (iv) comparison with direct neighbors to reach a consensus.



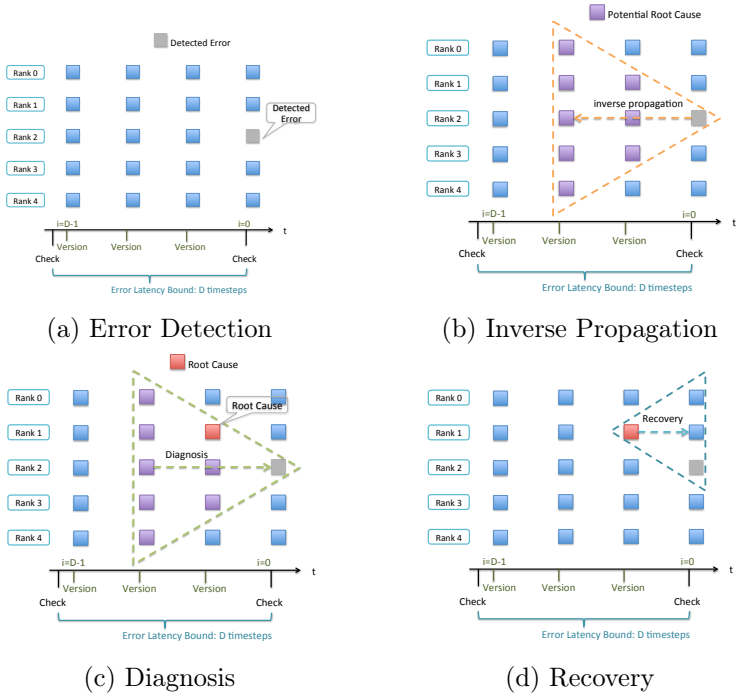
(a) 3-point 1D (b) 5-point 2D (c) 7-point 3D

Figure 2: Stencil patterns: an error propagates to direct neighbors (blue) in a timestep.

The interval between two consecutive error detections bounds the error latency. Given error location and timing, application logic and dataflow (see Figure 2) – is used to invert worst-case error propagation, identifying all data points in past that could have contributed to this error manifestation. These data points are called potential root causes (**PRC**). To bound error impact more precisely, PRCs can be tested (diagnosis), eliminating many of the initial PRCs (see Figure 3); for stencils, this can accomplished by recomputing intermediate states from versions (courtesy of GVR) and comparing to previously saved results. If the values match, the PRC can be pruned. At last, recovery is applied to the reduced set of PRCs and their downstream error propagation paths. In Section 4, we develop a model, quantifying the PRCs for a given error latency. It takes thousands of timesteps to corrupt even 1% of the data, but traditional CR assumes all application data is corrupted.

[1]Errors that cannot be detected are beyond the ability of any error recovery system to consider.

[2]Assuming expensive checks means that any improvements in checking can be incorporated – cost is not a disqualifier.



(a) Error Detection (b) Inverse Propagation

(c) Diagnosis (d) Recovery

Figure 3: ABFR in a 3-point 1D Stencil.

Explaining our example in detail (Figure 3), there are five ranks in the stencil computation. Each box in the figure represents the data of a rank. Each rank exchanges data with its two neighbors at each timestep, using the incoming data at the next step. At a certain timestep, an error is detected. Inverse propagation identifies all potential root causes (PRCs) of the error (purple boxes). Diagnosis of the PRCs eliminates most of them, leaving only viable one (the red box). Recovering the red box and its neighbors produces the correct application.

4 Analytical Performance Model

Suppose the stencil works on M elements, each updated every timestep. Every D timesteps, an error detector is invoked to examine the state of M elements. Therefore the error latency bound is D timesteps. Then, a version of the state is stored. For ABFR, additional versions of data are created every V timesteps between two error detections. In order to simplify the model, we make the following assumptions:

- Errors occur randomly in space and time.
- Only a single error occurs between two error detections.
- Only a single manifestation of the error is detected.

Note that these assumptions are commonly used to model CR. The impli-

Variable	Definitions	Units
M	Application size (number of elements × element size)	bytes
m	Box Size (number of elements in one box × element size)	bytes
n	Number of boxes assigned to one process	-
p	Number of processes in computation	-
t	Time to advance one element by one timestep	seconds/byte
d	Time to run the detector on one element	seconds/byte
s	Time to store one element (versioning)	seconds/byte
r	Time to reload one element	seconds/byte
c	Time to compare one element with a previous version	seconds/byte
D	Detection interval, Error Latency Bound	timesteps
V	Versioning interval	timesteps
α	Ratio of versioning interval to detection interval, $V = \alpha D$	-
B	Number of versions between two detections, $B = \frac{D}{V} = \frac{1}{\alpha}$	-
λ	Error rate	errors/(second*byte)
λM	System error rate	errors/second
$(1 - e^{-\lambda M})$	Probability of having an error in one second	-
E	Expected cost of completing computation of D timesteps	(cpu) seconds
Rec	Recovery cost: the amount of work required to recover	(cpu) seconds
T	Expected runtime of completing computation of D timesteps	seconds
$RecLat$	Recovery latency: runtime critical path for recovery	seconds

Table 1: Table of Notations

cations are as follows: since no other error can occur between two checks, only one recovery is needed (no error strikes during recovery). Although these assumptions cover most cases in practice, it is possible to extend the analysis to handle additional errors (see Section 6 for a discussion).

If an error is detected, we first identify the potential root causes based on stencil pattern. Let $step(j)$ be the number of additional elements that got corrupted after one timestep. This typically depends on the dimension of the grid, and the number of neighbors involved in the computation for one timestep. We define $root(i)$ as the number of potential root causes i timesteps ago and $AllRoot$ as the total number of potential root causes over the past D timesteps as follows:

$$root(i) = 1 + \sum_{j=1}^{i} step(j),\ AllRoot = \sum_{i=0}^{D-1} root(i)\ .$$

	1D	2D	3D
$step(i)$	2	$4i$	$4i^2 + 2$
$root(i)$	$2i + 1$	$2i^2 + 2i + 1$	$1 + \frac{4}{3}i^3 + 2i^2 + \frac{8}{3}i$
$AllRoot$	D^2	$\frac{2}{3}D^3 + \frac{1}{3}D$	$\frac{1}{3}D^4 + \frac{2}{3}D^2$

Table 2: Expressions for *step*, *root*, and *AllRoot* functions for 1, 2 and 3 dimensional grids, assuming an element interacts only with its direct neighbors.

Table 2 shows the expressions for *step*, *root* and *AllRoot* for 1D, 2D, and 3D stencils. We assume that diagnosis is done by recomputing elements from the last checked version, which was D timesteps ago. We can compare the result against intermediate versions. If the recomputed data differs from the version, then the error occurred between the last two versions. Note that with a version at every step, we can narrow the root cause of an error to a single point. Suppose the error occurred j timesteps ago, then the time required for diagnosis is the time to recompute, reload and check $(t + r + c)$ each element against the version from iteration $D - 1$ to j as illustrated in Figure 3c:

$$diag(i) = r \cdot root(D) + (t + r + c) \sum_{j=i}^{D-1} root(j) \ .$$

Once potential root causes are pruned, recovery is done by recomputing the reduced set of potential root causes and affected data from the last correct version, as illustrated in Figure 3d:

$$recomp(i) = -t + (t + s) \sum_{j=0}^{i} root(j) \ .$$

As discussed in Section 3, ABFR allows overlapping recovery. In that case, the recovery cost (work needed) is the critical metric. If recovery cannot be overlapped, then recovery latency (parallel time) is appropriate. We model both of these for 2D stencils. We refer the reader to the Appendix Section 8 for the analysis of 1D and 3D stencils.

4.1 Recovery Cost

We first quantify the total cost (amount of work due to computation, detection, versioning and recovery, counted in CPU time) of the ABFR approach, as a function of error rate λ (errors per second per byte) and detection interval D, denoted by $\mathbb{E}_{ABFR}$ and compare it with the classical CR (Checkpoint/Restart) approach, denoted by $\mathbb{E}_{CR}$.

Program execution is divided into equal-size segments of D timesteps. The time needed to complete one segment with p processes is $\frac{DtM}{p}$, and the total CPU time on computation is DtM. Similarly, we spend a total of dM time on detection and BsM time on versioning, where B is the number of versions taken between two detections For CR, we use $B = 1$, CR creates a version every D timesteps. Then, we assume that errors occur following an exponential distribution, and the probability of having an error during the execution of one segment is denoted by $1 - e^{-\lambda M \frac{DtM}{p}}$, where λM is the application error rate. Therefore, we can write $\mathbb{E}_{CR}$ and $\mathbb{E}_{ABFR}$ as functions

of D and λM as follows:

$$\mathbb{E}_{CR} = DtM + dM + sM + \left(1 - e^{-\lambda M \frac{DtM}{p}}\right) Rec_{CR} \ , \tag{1}$$

$$\mathbb{E}_{ABFR} = DtM + dM + BsM + \left(1 - e^{-\lambda M \frac{DtM}{p}}\right) Rec_{ABFR} \ . \tag{2}$$

The main difference between both approaches lies in recovery cost. Recovery of CR includes reloading data and full recomputation, while ABFR includes diagnosis cost, different data reloading, and reduced recomputation cost. For CR, we have directly:

$$Rec_{CR} = rM + DtM \ . \tag{3}$$

For ABFR, let $B = \frac{D}{V}$ denote the number of versions taken between two detections. We number versions backwards, from $j = 0$ (timestep 0) up to $j = B - 1$ (timestep $(B-1)V$). The last checked version (timestep D) has been versioned too ($j = B$). We introduce the notation $A(j)$, which is the total number of potential root causes between two versioned timesteps jV and $(j+1)V$, excluding $(j+1)V$ but including jV:

$$A(j) = \sum_{k=jV}^{(j+1)V-1} root(k) \ .$$

Therefore, $\frac{A(j)}{AllRoot}$ denote the probability that the error occurred between version j and $j+1$, and we can write:

$$Rec_{ABFR} = \sum_{j=0}^{B-1} \frac{A(j)}{AllRoot} \left(diag(j) + recomp(j)\right) \ .$$

The diagnosis is done by recomputing all potential root causes from timesteps $D-1$ up to version j, that is timestep jV. In addition, we need to pay $(r+c)root(kV)$ for every version k that passed the diagnosis test, that is from version $B-1$ to j included. Therefore, we can write:

$$diag(j) = r \cdot root(D) + t \sum_{k=jV}^{D-1} root(k) + (r+c) \sum_{k=j}^{B-1} root(kV) \ .$$

Because we may have gaps in-between versions, we do not know the exact location of the root cause of the error. Therefore, we recompute starting from version $j+1$ instead of j. We must recompute all potential affected elements from timestep $(j+1)V - 1$ to 0. At timestep $(j+1)V - 1$, there are $root((j+1)V-1)$ potential root causes elements to recompute. At every timestep, the number of elements to recompute increases by $step(j)$, so that

there are a total of $root(2(j+1)V)$ elements to recompute at timestep 0. Therefore, we can write:

$$recomp(j) = t \sum_{k=(j+1)V-1}^{2(j+1)V} root(k) + s \sum_{k=j+1}^{2(j+1)} root(kV) \ .$$

Finally, we obtain the recovery cost as a function of the detection interval D:

$$Rec_{ABFR} = \frac{8}{15} t(\alpha^5 - 5\alpha^3 + 9\alpha + 5)D^3 + O(D^2), \quad \text{where } \alpha = \frac{1}{B}. \tag{4}$$

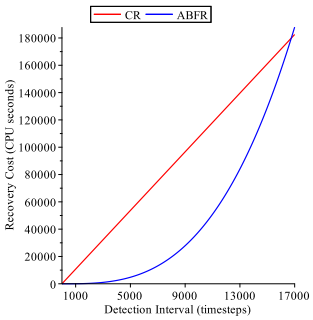


Figure 4: Recovery Cost vs. Detection Interval ($M = 32768^2, t = 10^{-8}, d = 100t, r = 10^{-9}, s = 10^{-8}, \alpha = \frac{1}{4}$)

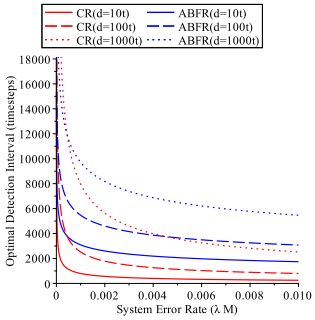


Figure 5: Optimal Detection Interval vs. Error Rate ($M = 32768^2, p = 4096, t = 10^{-8}, r = 10^{-9}, s = 10^{-8}, \alpha = \frac{1}{4}$)

Recovery Cost Comparison The dominant cost in recovery is recomputation. It is $O(DM)$ for CR in Equation 3 and $O(D^3)$ for ABFR in Equation 4. Suppose the number of elements in one dimension of stencil is U, we have $M = U$, $M = U^2$ and $M = U^3$ for 1D, 2D, and 3D stencil respectively. Since CR always recompute all the data, the corresponding recomputation cost is $O(DU)$, $O(DU^2)$ and $O(DU^3)$. In constrast, ABFR only need to recompute a small fraction of the M elements. The corresponding recomputation cost is $O(D^2)$, $O(D^3)$ and $O(D^4)$ respectively (see Appendix Section 8. Note that the detection interval D (or error latency) is much smaller than the number of elements in one dimension U.

We plot the recovery cost of CR and ABFR as a function of detection interval (error latency) in Figure 4 (note that CR creates 1 version during D timesteps, while ABFR creates B versions. The plot uses $B = \frac{1}{\alpha} = 4$). We observe that CR grows linearly with detection interval. While ABFR increases slowly for less than 9,000 and outperforms CR for error latencies up to 17,000 timesteps. This range of 1,000 to 17,000 time steps corresponds to 3 seconds to about 1 minute. After that, most data are corrupted, hence ABFR cannot further improve the performance by bounding error impact.

Let $H = \frac{\mathbb{E}}{DtM}$ denote the expected overhead with respect to the computation cost without errors. We have

$$\begin{aligned} H_{CR} &= 1 + \frac{d+s}{Dt} + \lambda M(rM + DtM), \\ H_{ABFR} &= 1 + \frac{b}{D} + \lambda M a D^3, \\ \text{where } a &= \frac{8}{15}t(\alpha^5 - 5\alpha^3 + 9\alpha + 5) \text{ and } b = \frac{\alpha d + s}{\alpha t} \ . \end{aligned} \tag{5}$$

Optimal Detection Interval Minimizing the overhead, we derive the following optimal detection interval for Checkpoint-Restart and ABFR:

$$D^*_{CR} = \sqrt{\frac{(d+s)p}{\lambda M^2 t^2}} \ , \text{ and } D^*_{ABFR} = \sqrt[4]{\frac{bp}{3a\lambda M}} \ . \tag{6}$$

Empirical studies of petascale systems have shown MTBF's of three hours at deployment [28], and allowing for the greater scale of exascale systems [10, 33], future HPC system MTBFs have been projected as low as 20 minutes [23]. To explore possibilities for a broad range of future systems (including cloud), we consider system error rates (errors/second) ranging from 0 (infinite MTBF) to 0.01 (1 minute MTBF). We assume the application runs on the entire system, setting λM to the system error rate.

We plot the optimal detection interval as a function of error rate λM in Figure 5. We observe that as error rate increases, the optimal detection interval of CR drops faster than ABFR for varied error detector cost, indicating CR demands more frequent error detection in high error rate environments. So, here the goal is to be lazy in error detection checking, because deep application-semantics are assumed to be expensive. Higher numbers for optimal detection interval are good. Plugging D^* back into H, we derive that

$$H^*_{CR} = 1 + 2M\sqrt{\frac{(d+s)}{p}}\sqrt{\lambda} + rM^2\lambda, \tag{7}$$

$$\text{and } H^*_{ABFR} = 1 + \frac{4}{3}\sqrt[4]{\frac{3ab^3\lambda M}{p}} \ . \tag{8}$$

We plot the overhead as a function of error rate, when using the optimal detection interval, in Figure 6. With growing error rates, CR incurs high overhead. In contrast, ABFR significantly reduces overhead and performs stable even for high error rates.

4.2 Recovery Latency

We model recovery latency (parallel execution runtime). Large-scale simulations overly decompose a grid into boxes, enabling parallelism and load

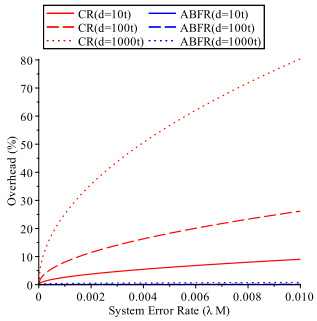


Figure 6: Overhead vs. Error Rate Using Optimal Detection Interval ($M = 32768^2, p = 4096, t = 10^{-8}, r = 10^{-9}, s = 10^{-8}, \alpha = \frac{1}{4}$)

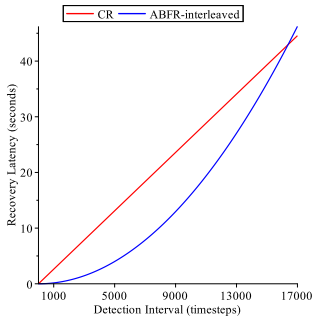


Figure 7: Recovery Latency vs. Detection Interval ($M = 32768^2, m = 65536, p = 4096, n = 4, t = 10^{-8}, d = 100t, r = 10^{-9}, s = 10^{-8}, \alpha = \frac{1}{4}$)

balance. As in Figure 8, each process is assigned a set of boxes; each of which is associated a halo of ghost cells. The square grid of $\sqrt{M} \times \sqrt{M}$ elements is partitioned into square boxes of size $\sqrt{m} \times \sqrt{m}$. We have $\frac{M}{m}$ boxes mapped on to p processes.

Recovery latency, *RecLat*, is determined by the process with the most work. For CR, we assume perfect load balance; each process has n boxes, so $npm = M$. Thus $RecLat_{CR}$ reloads n boxes and recomputes them for D timesteps:

$$RecLat_{CR} = n(rm + Dtm) \ . \tag{9}$$

For ABFR, recovery latency is determined by the process with the most corrupted boxes. For simplicity, we recompute entire box even it is partially corrupted in ABFR. In an ideal case, the actual corrupted boxes are owned by processes uniformly, making the number of corrupted box of each process, equal to $n_{ideal} = \frac{root(D)}{mp} = \frac{2D^2}{mp} + O(D)$. For the interleaved mapping (see Figure 8), there are $\sqrt{M/m}$ boxes in one row, so the vertical distance between two boxes assigned to the same rank is $\frac{p}{\sqrt{M/m}}$ (box). The length $2D$ is the range of error spread. The slowest process would have $n_{inter} = \frac{2D}{\sqrt{m}} / \frac{p}{\sqrt{M/m}} = \frac{2D\sqrt{M}}{mp}$ corrupted boxes. Then, for an error at step j, we have:

$$diag(j) = rm + t \sum_{k=jV}^{D-1} m + (r + c) \sum_{k=j}^{B-1} m,$$

$$recomp(j) = t \sum_{k=0}^{(j+1)V} m + s \sum_{k=0}^{j+1} m \ .$$

To compute the recovery latency Rec_{box} per box, we proceed as before:

$$\begin{aligned} Rec_{box} &= \sum_{j=0}^{B-1} \frac{A(j)}{AllRoot} \left(diag(j) + recomp(j)\right) \\ &= tm\alpha D + o(D). \end{aligned}$$

Multiplying Rec_{box} by the corresponding number of boxes in the ideal and interleaved scenarios, we obtain

$$RecLat_{ideal} = \frac{2t\alpha}{p} D^3 + O(D^2), \tag{10}$$

$$RecLat_{inter} = \frac{2t\alpha\sqrt{M}}{p} D^2 + O(D) \ . \tag{11}$$

Comparing Equations (9) and (10), we conclude that as long as the latency is not long enough to infect all assigned boxes of one process, ABFR would produce better performance. We plot $RecLat_{CR}$ and $RecLat_{inter}$ as a function of detection interval in Figure 7. Similar as in Figure 4, CR increases linearly with detection interval. And ABFR outperforms CR for the detection interval from 0 to 17,000 timesteps. But the gap between their recovery latencies is smaller compared with that in recovery cost. The gap between recovery latencies mainly depends on the difference in the number of boxes that the slowest process needs to work on. Therefore ABFR is at most $n = 4$ times better in the plot configuration.

Optimal Detection Interval. We derive the expected runtime of CR and ABFR to successfully compute D timesteps.

$$\begin{aligned} T_{CR} &= Dnmt + dnm + snm + (1 - e^{-\lambda MDnmt}) RecLat_{CR} \\ T_{ABFR} &= Dnmt + dnm + Bsnm + (1 - e^{-\lambda MDnmt}) RecLat_{ABFR} \end{aligned}$$

The overhead $H = \frac{T}{Dnmt}$ of CR and ABFR are given by

$$\begin{aligned} H_{CR} &= 1 + \frac{d+s}{Dt} + \lambda Mn(rm + Dtm), \\ H_{ideal} &= 1 + \frac{\alpha d + s}{\alpha Dt} + \lambda M \frac{2t\alpha}{p} D^3, \\ H_{inter} &= 1 + \frac{\alpha d + s}{\alpha Dt} + \lambda M \frac{2t\alpha\sqrt{M}}{p} D^2 \ . \end{aligned}$$

Minimizing the overhead, we derive the optimal detection interval for CR, ideal ABFR and interleaved ABFR, respectively, as follows:

$$D^*_{CR} = \sqrt{\frac{(d+s)p}{\lambda M^2 t^2}}, D^*_{ideal} = \sqrt[4]{\frac{(\alpha d + s)p}{6\alpha^2 t^2 \lambda M}}, D^*_{inter} = \sqrt[3]{\frac{(\alpha d + s)p}{4\alpha^2 \lambda M^{\frac{3}{2}} t^2}} \ .$$

The optimal interval D^*_{CR} of CR is the same as in Equation (6). The optimal interval for ideal-ABFR is $D^*_{ideal} = \Theta(\lambda^{\frac{-1}{4}})$, the same order of magnitude as D^*_{ABFR}, the optimal value of Equation (6) for the recovery cost. D^*_{inter} is different due to imbalanced recovery.

5 Model Validation: Experiments

5.1 Methodology

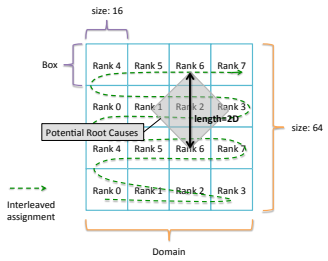


Figure 8: Interleaved domain decomposition

Number of ranks	4096
Domain size	10^9 (32768x32768)
Number of boxes	16384 (128x128)
Box size	65536 (256x256)
#Box per process	4

Table 3: Experiment Configurations

Workload We use Chombo 2D heat equation codes as the testbed to validate the model. Chombo [14] is a library that implements block-structured adaptive mesh refinement technique. The 2D heat equation codes, implemented with Chombo library, solve a parabolic partial differential equation that describes the distribution of heat in a given region over time. It is a 5-point 2D stencil program and deploys an interleaved domain decomposition method. An example of such decomposition for a 64x64 domain and 8 ranks is shown in Figure 8.

We enhanced Chombo with two recovery schemes – CR (baseline) and ABFR. The CR scheme saves a version in memory after each error check. When an error is detected, CR rolls back to the last checked version and recomputes. Note that it is a improved version of classical CR because it avoids iteratively rollback and recompute until the error is corrected. ABFR creates 3 additional versions between two error checks, i.e. 4 versioning intervals in 1 detection interval. In recovery, ABFR diagnoses potential root causes using application knowledge and intermediate versions, then only recomputes corrupted data.

Experiment Design We explore the performance of CR and ABFR for varied error detection intervals and error latencies. The configuration of experiments is listed in Table 3. We run 4,096 ranks and solve the heat equation for a domain of 10^9 elements. With this problem size, we vary the detection interval from 1,000 timesteps to 13,000 timesteps, producing potential corrupted data fractions that range from 0.2% to 32%. ABFR always

creates 4 versions, the interval between versions increases with the detection interval. For each detection interval, we sample error latencies uniformly, injecting an error in each versioning interval. We measure the performance for each error latency and calculate the average results to produce performance for the detection interval length.

All experiments were conducted on Edison, the Cray XC30 at NERSC (5576 nodes, dual 12-core Intel IvyBridge 2.4 GHz, 64GB memory). We use 4,096 ranks, typically spread over 342 nodes. The results are average of three trials.

Metrics We use metrics – *recovery cost*, *recovery latency* and *data read* (IO) for comparison. *Recovery cost* is the total amount of work (CPU time) required to recover. *Recovery latency* is the runtime critical path for application recovery. *Data read* is the amount of data restored during recovery, representing I/O cost.

5.2 Results

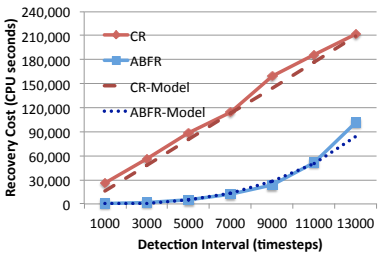


Figure 9: Recovery Cost vs. Detection Interval (Model plotted for experiment configuration and measured $t = 1.5 * 10^{-8} second$)

Recovery Cost Figure 9 plots the recovery cost for varied detection intervals (1000 to 13,000 timesteps). Recovery cost for CR grows linearly with detection interval (error latency). The recovery cost of ABFR is initially 400x lower (62 vs. 25,700 CPU seconds at 1000 timesteps), and it grows slowly. The gap between them increases steadily but the ratio decreases. Even at 13000 timesteps, ABFR has 2x lower recovery cost. In contrast to CR, ABFR effectively focuses recovery effort, using diagnosis to reduce cost.

Figure 9 also plots the performance model (dotted and dash lines), showing a close match (for broader comparison see Figure 4). As expected, ABFR cost starts lower and grows polynomially with the detection interval.

Recovery Latency Figure 10 compares the recovery latency with a range of detection intervals. For shorter intervals, ABFR has much lower recovery latency, reducing recovery latency by up to 4x (detection interval of 1000 timesteps). The recovery latency is determined by the slowest process. Each process in CR recomputes all 4 boxes assigned to it at every timestep.

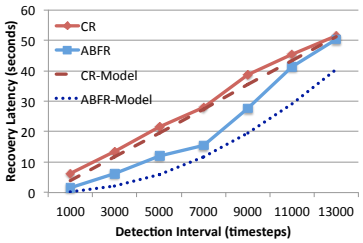


Figure 10: Recovery Latency vs. Detection Interval (Model plotted for experiment configuration and measured $t = 1.5 * 10^{-8} second$)

While in ABFR, for 1,000 time steps, only 41 boxes are identified potentially corrupted, so processes involved in recovery work at most on one box, producing 4x better performance. As detection interval increases, the error may propagate to larger area, making it more likely that each process has more boxes to handle. At detection interval (error latency) of 13,000 timesteps, ABFR has same performance as CR.

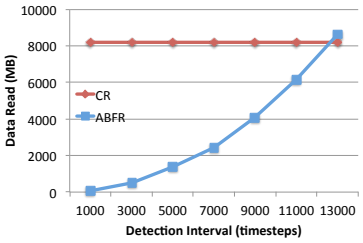


Figure 11: Data Read (MB) vs. Detection Interval

The dotted and dash lines in Figure 10 are performance model results using parameter values of our experiments (see also Figure 7). Our experiment results have similar curves as the model. The recovery latency of CR grows almost linearly with detection intervals. While ABFR produces low recovery latencies for short detection intervals and then chases up with CR with expanding detection intervals. The measured ABFR performance are slightly worse than the model because we only keep the highest order terms in the model for simplification but omit some other costs.

Data Read (IO) An important cost for recovery is the reading of stored version data from the IO system. Figure 11 presents the data read versus detection intervals. In general, the data read increases with detection interval as on average the actual error latency is greater, causing ABFR to read parts of more versions. In contrast, CR always reloads the entire grid. Because ABFR intelligently bounds the error impact and loads the required data to recover all potential errors, it reduces data read by as much as 1000-fold.

6 Discussion

Generality of ABFR As a type of ABFT, ABFR requires sufficient application knowledge to design inverse error propagation, diagnose and focus recovery. However, this knowledge can be coarse-grained. Our studies show that ABFR is helpful for several classes of applications. Applications that have regular data dependencies, such as stencils and adaptive mesh refinement (AMR) can easily adopt ABFR to bound error effect and confine recovery. Some applications have dependency tables or graphs that can be exploited by ABFR. Such examples include broad graph processing algorithms and task-parallel applications. Some applications have properties that limit the spread of errors. For instance, N-Body tree codes have numerical cutoff that confine erroneous regions to some subtrees. Monte Carlo applications do not propagate errors across sampled batches. We plan to extend ABFR to these applications in future work.

Multiple Errors For simplicity we only model single errors. This assumption is common and underlies much of CR practice. There are several potential avenues for extension. First, multiple errors within a detection interval could trigger multiple ABFR responses. Alternatively, diagnosis could be extended to deal with multiple errors at once. These are promising directions for future work.

7 Related Work

Soft errors and data corruption for extreme-scale systems have been the target of numerous studies. A considerable number of researchers have already looked at error vulnerability. Some focus on error detection but rely on other methods to recover. Others work on designing recovery techniques. We classify related work into three categories: system-level resilience, ABFT (Algorithm-based Fault Tolerance) techniques and resilience for stencils.

System-level Resilience With the growing error rates, it has been recognized that single checkpoint cannot handle latent errors, as the rising frequency shrinks the optimal checkpoint interval [15], increasing the incidence of escaped errors. To address this reality at extreme scale, researchers have proposed multi-level checkpointing systems and multiple checkpoint-restart (MCR) approaches [5, 25, 30]. Such systems exploit fast storage (DRAM, NVRAM) to reduce I/O cost and keep multiple checkpoints around. Inexpensive but less-resilient checkpoints are kept in fast, volatile storage, and expensive but most-resilient checkpoints in parallel file system. When a latent error is detected, applications must search these checkpoints, attempting to find one that doesn't contain latent errors. The typical algorithm is to start from the more recent checkpoint, reexecute, then see if the latent error recurs. If it does, repeat with the next older checkpoint. This blind

search and global recovery incurs high overhead especially in case of errors with long latency, making MCR unsuitable for high error rates. In contrast, our ABFR approach exploits application-knowledge to narrow down the corrupted state, and only recompute that.

Algorithm-Based Fault-Tolerance Huang and Abraham [26] proposed a checksum-based ABFT for linear algebra kernels to detect, locate and correct single error in matrix operations. Other researchers extended Huang and Abraham's work for other specialized linear system algorithms, such as PCG for sparse linear system [31], dense matrix factorization [17], Krylov subspace iterative methods [11]. Below we address ABFT methods for stencils. Our work is similar to ABFT approaches, exploiting application knowledge for error detection, but adding the use of application knowledge to diagnose what state is potentially corrupt, and using that knowledge to limit recomputation, and thereby achieve efficient recovery from latent errors.

Resilience for Stencil Computations Researchers have explored error detection in stencil computations, for example exploiting the smoothness of the evolution of a particular dataset in the iterative methods to detect errors. Berrocal et al. [8] showed that an interval of normal values for the evolution of the datasets can be predicted, therefore any errors that make the corrupted data point outside the interval can be detected. Benson et al. [6] proposed a error detection method by using an cheap auxiliary algorithm/method to repeat the computation at the same time with original algorithm, and compare the difference with the results produced by the original algorithm. These work relied on Checkpoint-Restart to correct errors. Our ABFR approach can benefit from these efforts on application error checks.

Other studies have also explored resilience approaches for stencils. Gamell et al. [24] studied the feasibility of local recovery for stencil-based parallel applications. When a failure occurs, only the failed process is substituted with a spare one and rollbacks to the last saved state for the failed process and resumes computation. The rest of the domain continues communication. This technique assumes immediate error detection. Sharma et al. [32] proposed an error detection method for stencil-based applications using the predicted values by a regression model. Dubey et al. [18] explored local recovery schemes for applications using structured adaptive mesh refinement (AMR). Their work studied customizing resilience strategy exploiting the inherent structure and granularities within applications. Recovery granularities can be controlled at cell, box, and level (a union of boxes) for AMR depending on failure modes. This work also assumes immediate error detection. We share the context of stencils and attempts to confine error recovery scope, but our work is clearly different with its focus on latent errors.

8 Summary and Future Work

We propose an application-based focused recovery (ABFR) for stencil computations to efficiently recover from latent errors. This approach exploits stencil semantics and inexpensive versioned states to bound error impact and confine recovery scope. This focused recovery approach can yield significant performance benefits. We analyze and characterize the ABFR approach on stencils, creating a performance model parameterized by error rate and detection interval (error latency). Experiments with the Chombo heat equation application show promising results, reducing both recovery cost (up to 400x) and recovery latency (up to 4x), and validating the model. Future directions include experiments that extend ABFR ideas to other applications, and the analytical study of optimal versioning intervals and detection intervals.

References

[1] Juqueen. http://www.fz-juelich.de/ias/jsc/EN/Expertise/Supercomputers/JUQUEEN/JUQUEEN_node.html.

[2] Nersc cori. https://www.nersc.gov/users/computational-systems/cori/.

[3] S. Amarasinghe et al. Exascale software study: Software challenges in extreme scale systems. Technical report, DARPA IPTO, Air Force Research Labs, Tech. Rep, 2009.

[4] G. Aupy, A. Benoit, T. Hérault, Y. Robert, F. Vivien, and D. Zaidouni. On the combination of silent error detection and checkpointing. In *2013 IEEE 19th Pacific Rim International Symposium on Dependable Computing*, pages 11–20, Dec 2013.

[5] L. Bautista-Gomez et al. Fti: High performance fault tolerance interface for hybrid systems. In *SC '11*, 2011.

[6] A. R. Benson et al. Silent error detection in numerical time-stepping schemes. *Int. J. High Performance Computing Applications*, 2014.

[7] K. Bergman et al. Exascale computing study: Technology challenges in achieving exascale systems. Technical Report TR-2008-13, Defense Advanced Research Projects Agency Information Processing Techniques Office (DARPA IPTO), 2008.

[8] E. Berrocal et al. Lightweight silent data corruption detection based on runtime data analysis for hpc applications. In *HPDC '15*, 2015.

[9] F. Cappello. Fault tolerance in petascale/exascale systems: Current knowledge, challenges and research opportunities. *Int. J. High Performance Computing Applications*, 2009.

[10] F. Cappello et al. Toward exascale resilience: 2014 update. *Supercomput. Front. Innov*, 2014.

[11] Z. Chen. Online-abft: An online algorithm based fault tolerance scheme for soft error detection in iterative methods. In *PPoPP '13*, 2013.

[12] A. Chien, , et al. Versioned distributed arrays for resilience in scientific applications: Global view resilience. *Procedia Computer Science*, 2015.

[13] A. Chien et al. Exploring versioned distributed arrays for resilience in scientific applications: global view resilience. *Int. J. High Performance Computing Applications*, 2016.

[14] P. Colella et al. Chombo software package for AMR applications design document. Technical report, LBNL, 2009.

[15] J. T. Daly. A higher order estimate of the optimum checkpoint interval for restart dumps. *Future Gener. Comput. Syst.*, 2006.

[16] K. Datta et al. Optimization and performance modeling of stencil computations on modern microprocessors. *SIAM Rev.*, 2009.

[17] P. Du et al. Algorithm-based fault tolerance for dense matrix factorizations. In *PPoPP '12*, 2012.

[18] A. Dubey et al. Granularity and the cost of error recovery in resilient amr scientific applications. In *SC '16*, 2016.

[19] N. Dun et al. Data decomposition in monte carlo neutron transport simulations using global view arrays. *Int. J. High Performance Computing Applications*, 2015.

[20] N. Dun et al. Multi-versioning performance opportunities in bgas system for resilience. In *Int. Conf. High Performance Computing*. Springer, 2016.

[21] J. F. Epperson. *An introduction to numerical methods and analysis*. John Wiley & Sons, 2013.

[22] A. Fang and A. A. Chien. Applying gvr to molecular dynamics: Enabling resilience for scientific computations. Technical Report TR-2014-04, University of Chicago, 2014.

[23] K. Ferreira et al. Evaluating the viability of process replication reliability for exascale systems. In *SC'11*, 2011.

[24] M. Gamell et al. Local recovery and failure masking for stencil-based applications at extreme scales. In *SC '15*, 2015.

[25] E. Gelenbe. A model of roll-back recovery with multiple checkpoints. In *Proc. 2nd Int. Conf. on Software Engineering*, 1976.

[26] K.-H. Huang and J. Abraham. Algorithm-based fault tolerance for matrix operations. *IEEE Trans. Computers*, 1984.

[27] G. Lu et al. When is multi-version checkpointing needed? In *FTXS '13*, 2013.

[28] C. Martino et al. Lessons learned from the analysis of system failures at petascale: The case of blue waters. In *DSN '14*, 2014.

[29] C. D. Martino et al. Measuring and understanding extreme-scale application resilience: A field study of 5,000,000 hpc application runs. In *DSN '15*, 2015.

[30] A. Moody et al. Design, modeling, and evaluation of a scalable multi-level checkpointing system. In *SC '10*, 2010.

[31] M. Shantharam et al. Fault tolerant preconditioned conjugate gradient for sparse linear system solution. In *ICS '12*, 2012.

[32] V. C. Sharma, G. Gopalakrishnan, and G. Bronevetsky. Detecting soft errors in stencil based computations. *Geophysics*, 1983.

[33] M. Snir et al. Addressing failures in exascale computing. *Int. J. High Performance Computing Applications*, 2014.

A Extended analysis

In this section, we derive the optimal values for D and V that minimize the total expected execution time of the application for different scenarios. In the first scenario, we do not take advantage of the known error propagation pattern. We focus on the standard checkpoint and recovery approach and we derive the optimal D following the approach of Young/Daly. Then, in order to take full advantage of the known error propagation pattern, we focus on the simple scenario where $V = 1$, which allows us to cut down the recomputation time in case of error. Ultimately, we move to the general scenario with arbitrary values for V. We must find a tradeoff between the amount of time spent versioning vs recomputing upon error, and we derive optimal values for both D and V.

A.1 Standard checkpoint and recovery

In this section, we set $V = D$, so that we only version after a successful error check. When an error is detected, we simply reload the last correct version, and we recompute all D iterations from there. Let $\mathbb{E}$ denote the expected time needed to execute D iterations successfully. We first pay DtM, the cost for executing D iterations, where t is the time needed to compute a single element. Then, we pay dM, the cost for running the detector on the M elements. With probability $1 - e^{-\lambda M \frac{DtM}{p}}$, there was an error and we pay rM, the time needed to recover from the last correct version and DtM, the time needed to recompute all elements. With probability $e^{-\lambda M \frac{DtM}{p}}$, there was no error and we are done. Finally, in both cases we must store the correct version with cost sM. Therefore, we can write:

$$\mathbb{E} = DtM + dM + \left(1 - e^{-\lambda M \frac{DtM}{p}}\right)(rM + DtM) + sM \ .$$

Then, let $H = \frac{E}{DtM}$ denote the expected overhead with respect to the execution time without errors (DtM). Using Taylor series to approximate $\left(e^{-\lambda M \frac{DtM}{p}}\right)$ to $1 - \lambda M \frac{DtM}{p}$, and keeping only first order terms, we can write:

$$H = 1 + \frac{d+s}{Dt} + \lambda M(rM + \frac{DtM}{p}) + O(\lambda^2) \ .$$

Finally, differentiating H and solving for D, we derive that:

$$D^* = \sqrt{\frac{(d+s)p}{\lambda M^2 t^2}} \ ,$$

hence we have $D^* = O(\lambda^{-\frac{1}{2}})$. Note that this result holds for any grid dimension.

A.2 Version every step

In this section, we set $V = 1$, so that a version is taken after every iteration. Let $\mathbb{E}$ denote the expected time needed to execute D iterations successfully. We first pay DtM, the cost for executing D iterations, where t is the time needed to compute a single element. Then, we pay DsM, the cost for versioning at every step, where s denote the time needed to store a single element. Finally, we pay dM, the cost for running the detector on the M elements. With probability $1 - e^{-\lambda M \frac{DtM}{p}}$, there was an error and we pay Rec, the expected time needed to trace the source of the error from the single manifestation and to recompute all corrupted elements from there.

$$\mathbb{E} = DtM + DsM + dM + \left(1 - e^{-\lambda M \frac{DtM}{p}}\right) Rec \ .$$

We assume that the error was detected at iteration 0 and we number iterations backwards, from $i = 0$ to $i = D-1$. $i = D$ corresponds to the last check. With probability $\mathbb{P}_{err}^{line}(i) = \frac{root(i)}{AllRoot}$, the error occurred i iterations ago. Let $diag(i)$ denote the time needed to find the root cause of the error, from iteration $D-1$ to i, and let $recomp(i)$ denote the time needed to recompute all corrupted elements, from iteration $i-1$ to 0. Accounting for all possible scenarios, we can write:

$$Rec = \sum_{i=0}^{D-1} \mathbb{P}_{err}^{line}(i) \left(diag(i) + recomp(i)\right) \ .$$

Diagnosis is done by recomputing all potential root causes from iteration $D-1$ to i, and by comparing the result with the corresponding versions. First, we pay $r.root(D)$ in order to reload the last correct version. Then, for each iteration j, we must pay $r.root(j)$ to reload the corresponding version, $t.root(j)$ to recompute all the potential root causes at iteration j, and finally $c.root(j)$, the cost to compare the data against the version. Altogether, we can write:

$$diag(i) = r.root(D) + (t + r + c) \sum_{j=i}^{D-1} root(j) \ .$$

When the diagnosis is done, we have to account for the recomputation cost. The number of elements to recompute grows linearly from i downto 0. Indeed, there is only one element to recompute at iteration i, or $root(0)$, and there are $root(i)$ elements to recompute at iteration 0. Also, the root cause of the error itself at iteration i has been corrected during the diagnosis. Therefore, we need to pay $(t + s)root(j)$, the cost to recompute and store each corrupted elements with j from 0 to i, to which we remove $t.root(0)$ and we can write:

$$recomp(i) = -t + (t + s) \sum_{j=0}^{i} root(j) \ .$$

Then,

$$Rec = \sum_{i=0}^{D-1} \frac{root(i)}{AllRoot} \left(-t + (r + t + c) \sum_{j=i}^{D-1} root(i) + (t + s) \sum_{j=0}^{i-1} root(i) \right) \ .$$

Note that $AllRoot = \sum_{j=0}^{D-1} root(i)$, so that we can extract t and rewrite Rec as follows:

$$Rec = t(AllRoot - 1) + \sum_{i=0}^{D-1} \frac{root_i}{AllRoot} \left((r + c) \sum_{j=i}^{D-1} root(i) + s \sum_{j=0}^{i-1} root(i) \right) \ .$$

Finally, let $H = \frac{E}{DtM}$ denote the expected overhead with respect to the execution time without errors (DtM). Using Taylor series to approximate $\left(1 - e^{-\lambda M \frac{DtM}{p}}\right)$ to $\lambda M \frac{DtM}{p}$, keeping only first order terms, we can write:

$$H = 1 + \frac{s}{t} + \frac{d}{Dt} + \frac{\lambda M}{p} Rec + O(\lambda^2) \ .$$

Instantiating H with the correct $step(i)$ function, differentiating and solving for D, we can derive the optimal detection interval D^*.

1D case. We set $step(i) = 2$, and we get:

$$Rec = t(D^2 - 1) + \sum_{i=0}^{D-1} \frac{2i+1}{D^2} \left((r+c)(D^2 - i^2) + s(i^2 - 1)\right) \ .$$

Keeping only terms in D^2, we get:

$$Rec = D^2 \frac{c + r + s + 2t}{2} + O(D) \ .$$

So that:

$$H = 1 + \frac{s}{t} + \frac{d}{Dt} + \frac{\lambda M}{p} D^2 \left(\frac{c + r + s + 2t}{2}\right) + O(\lambda^2 D) \ .$$

Differentiating and solving for D we get:

$$D^* = \sqrt[3]{\frac{dp}{\lambda M t (c + r + s + 2t)}} \ ,$$

hence we have $D^* = O(\lambda^{-\frac{1}{3}})$.

2D case. Similarly, we set $step(i) = 4i$, and we get:

$$Rec = D^3 \frac{c + r + s + 2t}{3} + O(D^2) \ .$$

Therefore, we can write:

$$H = 1 + \frac{s}{t} + \frac{d}{Dt} + \frac{\lambda M}{p} D^3 \left(\frac{c + r + s + 2t}{3}\right) + O(\lambda^2 D^2) \ .$$

Then, differentiating and solving for D we get:

$$D^* = \sqrt[4]{\frac{dp}{\lambda M t (c + r + s + 2t)}} \ ,$$

and we have $D^* = O(\lambda^{-\frac{1}{4}})$.

3D case. Let $step(i) = 4i^2 + 2$, we derive that:

$$Rec = D^4 \frac{c + r + s + 2t}{6} + O(D^3) \ .$$

Therefore, we can get:

$$H = 1 + \frac{s}{t} + \frac{d}{Dt} + \frac{\lambda M}{p} D^4 \left(\frac{c + r + s + 2t}{6} \right) + O(\lambda^2 D^3) \ ,$$

and finally differentiating and solving for D we get:

$$D^* = \sqrt[5]{\frac{dp}{\lambda M t (c + r + s + 2t)}} \ ,$$

with $D^* = O(\lambda^{-\frac{1}{5}})$.

A.3 Version at a given interval

In this section, we consider the general case, and V can be anywhere between 1 and D. Let $B = \frac{D}{V}$ denote the number of versions taken between two detections. As before, we denote by $\mathbb{E}$ the expected time needed to successfully execute D iterations. We pay DtM, the cost for executing tM elements for D iterations, BsM, the cost of storing B versions, and dM, the cost of running the detector. With probability $1 - e^{-\lambda M \frac{DtM}{p}}$ there was an error, and we need to recover from the last correct version. Therefore, we can write:

$$\mathbb{E} = DtM + BsM + dM + \left(1 - e^{-\lambda M \frac{DtM}{p}}\right) Rec \ .$$

Similarly as for iterations, we number versions backwards, from $j = 0$ (iteration 0) up to $j = B - 1$ (iteration $(B-1)V$). The last checked version (iteration D) has been versioned too ($j = B$).

We introduce the notation $A(j)$, which is the total number of potential root causes between two versioned iterations jV and $(j+1)V$, excluding $(j+1)V$ but including jV:

$$A(j) = \sum_{k=jV}^{(j+1)V-1} root(k) \ .$$

Then, let $\mathbb{P}_{err}^{area}(j) = \frac{A(j)}{AllRoot}$ denote the probability that the error occurred between version j and $j+1$. We can write:

$$Rec = \sum_{j=0}^{B-1} \frac{A(j)}{AllRoot} \left(diag(j) + recomp(j) \right) \ .$$

The diagnosis is done by recomputing all potential root causes from iterations $D-1$ up to version j, that is iteration jV. In addition, we need to pay $(r+c)root(kV)$ for every version k that passed the check, that is from version $B-1$ to j included. Therefore, we can write:

$$diag(j) = t \sum_{k=jV}^{D-1} root(k) + (r+c) \sum_{k=j}^{B-1} root(kV) \ .$$

Then, recompute all corrupted elements. Because we have gaps in-between versions, we do not know the exact location of the root cause of the error. Therefore, we recompute starting from version $j+1$ instead of j. We must recompute all potential affected elements from iteration $(j+1)V-1$ to 0. At iteration $(j+1)V-1$, there are $root((j+1)V-1)$ potential root causes elements to recompute. At every iteration, the number of elements to recompute increases by $step(j)$, so that there are a total of $root(2(j+1)V)$ elements to recompute at iteration 0. Therefore, we can write:

$$recomp(j) = -t.root((j+1)V) + t \sum_{k=(j+1)V}^{2(j+1)V} root(k) + s \sum_{k=j+1}^{2(j+1)} root(kV) \ .$$

Now, let $H = \frac{E}{DtM}$ denote the expected overhead with respect to the execution time without errors (DtM). Using Taylor series to approximate $\left(1 - e^{-\lambda M \frac{DtM}{p}}\right)$ to $\lambda M \frac{DtM}{p}$, keeping only first order terms, we can write:

$$H = 1 + \frac{s}{tV} + \frac{d}{Dt} + \frac{\lambda M}{p} Rec + O(\lambda^2) \ .$$

Then, let $V = \alpha D$, where α is a fraction of D, so that we can write:

$$H = 1 + \frac{s + \alpha d}{\alpha t} \frac{1}{D} + \frac{\lambda M}{p} Rec + O(\lambda^2) \ .$$

Setting $b = \frac{s+\alpha d}{\alpha t}$, we can write:

$$H = 1 + \frac{b}{D} + \frac{\lambda M}{p} Rec + O(\lambda^2) \ .$$

1D case. In order to derive the optimal detection interval D^* and the optimal version interval V^*, we first set $V = \alpha D$, where $0 < \alpha \leq 1$, so that we have $V = O(D)$. For the 1D case, we set $step(i) = 2$, and keeping leading terms with respect to D, we get:

$$Rec = tD^2 \frac{2}{3} \left(3 - \alpha^3 + 4\alpha\right) + O(D) \ .$$

Then let $a = t\frac{2}{3}\left(3 - \alpha^3 + 4\alpha\right)$, so that we can rewrite H as follows:

$$H = 1 + \frac{b}{D} + \frac{\lambda M}{p} a D^2 + O(\lambda^2 D) \ .$$

Differentiating H with respect to D, and then solving for D, we can derive:

$$D^* = \sqrt[3]{\frac{1}{2}\frac{bp}{a\lambda M}} \ ,$$

hence we have $D^* = O(\lambda^{-\frac{1}{3}})$. Plugging D^* back into H, we derive that:

$$H^* = 1 + \frac{3}{2}\sqrt[3]{\frac{aM\lambda 2b^2}{p}}$$

Finally, in order to derive V^*, we must find α^*. Differentiating H with respect to α, we need to solve:

$$\frac{2}{3}p\frac{(\alpha d + s)\left(4\alpha^2 d - 3\alpha^4 d - \alpha^3 s - 4\alpha s - 6s\right)}{t\alpha^3} = 0 \ ,$$

which has to be done numerically.

2D case. Similarly, let $step(i) = 4i$, we derive that:

$$Rec = \frac{8}{15} t(\alpha^5 - 5\alpha^3 + 9\alpha + 5) D^3 + O(D^2) \ .$$

Then let $a = \frac{8}{15} t(\alpha^5 - 5\alpha^3 + 9\alpha + 5)$, so that we can rewrite H as follows:

$$H = 1 + \frac{b}{D} + \frac{\lambda M}{p} a D^3 + O(D^2) \ .$$

Differentiating H with respect to D and optimizing for D we can derive:

$$D^* = \sqrt[4]{\frac{1}{3}\frac{bp}{a\lambda M}} \ .$$

Plugging D^* back into H, we derive that:

$$H^* = 1 + \frac{4}{3}\sqrt[4]{\frac{3ab^3\lambda M}{p}} \ .$$

As for the 1D case, the optimal detection interval V^* has to be computed numerically.

3D case. Similarly, let $step(i) = 4i^2 + 2$, we derive that:

$$Rec = \frac{8}{315}t(140\alpha^5 - 23\alpha^7 - 252\alpha^3 + 250\alpha + 105)D^4 + O(D^3) \ .$$

Then let $a = \frac{8}{315}t(140\alpha^5 - 23\alpha^7 - 252\alpha^3 + 250\alpha + 105)$, so that we can rewrite H as follows:

$$H = 1 + \frac{b}{D} + \frac{\lambda M}{p}aD^4 + O(D^3) \ .$$

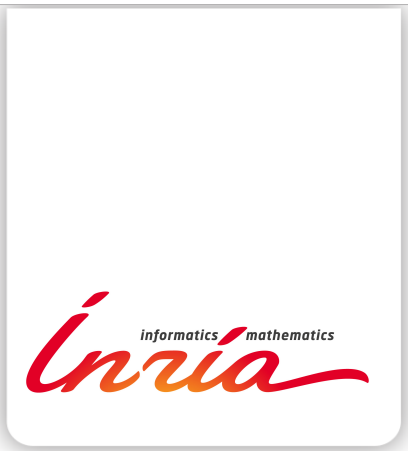
Differentiating H with respect to D and optimizing for D we can derive:

$$D^* = \sqrt[5]{\frac{1}{4}\frac{bp}{a\lambda M}} \ .$$

Plugging D^* back into H, we derive that:

$$H^* = 1 + \frac{5}{4}\sqrt[5]{\frac{4ab^4\lambda M}{p}} \ .$$

As for the 1D and 2D case, the optimal detection interval V^* has to be computed numerically.



RESEARCH CENTRE
GRENOBLE – RHÔNE-ALPES

Inovallée

655 avenue de l'Europe Montbonnot

38334 Saint Ismier Cedex

Publisher
Inria
Domaine de Voluceau - Rocquencourt
BP 105 - 78153 Le Chesnay Cedex
inria.fr

ISSN 0249-6399